Reference Capabilities for Concurrency Control*

Elias Castegren[1] and Tobias Wrigstad[2]

1 **Uppsala University, Sweden,** `Elias.Castegren@it.uu.se`
2 **Uppsala University, Sweden,** `Tobias.Wrigstad@it.uu.se`

Abstract

The proliferation of shared mutable state in object-oriented programming complicates software development as two seemingly unrelated operations may interact via an alias and produce unexpected results. In concurrent programming this manifests itself as data-races. Concurrent object-oriented programming further suffers from the fact that code that warrants synchronisation cannot easily be distinguished from code that does not. The burden is placed solely on the programmer to reason about alias freedom, sharing across threads and side-effects to deduce where and when to apply concurrency control, without inadvertently blocking parallelism.

This paper presents a reference capability approach to concurrent and parallel object-oriented programming where all uses of aliases are guaranteed to be data-race free. The static type of an alias describes its possible sharing without using explicit ownership or effect annotations. Type information can express non-interfering deterministic parallelism without dynamic concurrency control, thread-locality, lock-based schemes, and guarded-by relations giving multi-object atomicity to nested data structures. Unification of capabilities and traits allows trait-based reuse across multiple concurrency scenarios with minimal code duplication. The resulting system brings together features from a wide range of prior work in a unified way.

1998 ACM Subject Classification D.3.3 [Programming Languages] Language Constructs and Features – Classes and Objects

Keywords and phrases Type systems, Capabilities, Traits, Concurrency, Object-Oriented

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.5

1 Introduction

Shared mutable state is ubiquitous in object-oriented programming. Sharing can be more efficient than copying, especially when large data structures are involved, but with great power comes great responsibility: unless sharing is carefully maintained, changes through a reference might propagate unexpectedly, objects may be observed in an inconsistent state, and conflicting constraints on shared data may inadvertently invalidate invariants, etc. [28].

Multicore programming stresses proper control of sharing to avoid interference or data-races[1] and to synchronise operations on objects so that their changes appear atomic to the system. Concurrency control is a delicate balance: locking too little opens up for the aforementioned problems. Locking too much loses parallelism and decreases performance.

For example, parallelism often involves using multiple threads to run many tasks simultaneously without any concurrency control. This requires establishing non-interference by considering all the objects accessed by the tasks at any level of indirection.

* This work was partially funded by the Swedish Research Council project Structured Aliasing, the EU project FP7-612985 Upscale (`http://www.upscale-project.eu`), and the Uppsala Programming Multicore Architectures Research Centre (UPMARC)

[1] Two concurrent operations accessing the same location (read–write or write–write) without any synchronisation is a data-race. Non-interference allows only read–read races and no locks.

Mainstream programming languages place the burden of maintaining non-interference, acquiring and releasing locks, reasoning about sharing, etc. completely on the (expert) programmer. This is unreasonable, especially considering the increasing amount of parallelism and concurrency in applications in the age of multicore and manycore machines [6].

In this paper, we explore a reference capability approach to sharing objects across threads. A capability [31, 33] is a token that grants access to a particular resource, in our case objects. Capabilities present an alternative approach to tracking and propagating computational effects to check interference: capabilities assume exclusive access to their governed resources, or only permit reading. Thus, holding a capability implies the ability to use it fully without fear of data-races. This shifts reasoning from use-site of a reference to its creation-site.

We propose a language design that integrates capabilities with traits [39], *i.e.,* reusable units from which classes are constructed. This allows static checking at a higher level of abstraction than *e.g.,* annotations on individual methods. A *mode* annotation on the trait controls how exclusivity is guaranteed, *e.g.,* by completely static means such as controlling how an object may be referenced, or dynamically, by automatically wrapping operations in locks. A trait can be combined with different modes to form different capabilities according to the desired semantics: thread-local objects, immutable objects, unsharable linear objects, sharable objects with built-in concurrency control, or sharable objects for which locks must be acquired explicitly. This extends the reusability of traits across concurrency scenarios.

The sharing or non-sharing of a value is visible statically through its type. Types are formed by composing capabilities. Composition operators control how the capabilities of a type may share data, which ultimately controls whether an object can be aliased in ways that allow manipulation in parallel. Hiding a type's capabilities allows changing its aliasing restrictions. For example, hiding all mutating capabilities creates a temporarily immutable object which is consequently safe to share across threads (*cf.,* [9]).

Ultimately, with a small set of primitives—differently moded capabilities and composition operators—working in concert, the resulting system brings together many features from prior work: linear types [41, 23] and unique references [27, 34, 8, 17], regions [25], ownership types [16], universe types [22] and (fractional) permissions [9, 42]. As far as the authors are aware, there is no other *single* system that can express all of these concepts in a unified way.

This paper makes several contributions to the area of type-driven concurrency control:

- We present a framework for defining capabilities which work in concert to express a wide variety of concepts from prior work on alias control. The novel integration of capabilities with traits extends trait-based reuse across different concurrency scenarios without code duplication. Traits are guaranteed to be data-race free or free from any interference, which simplifies their implementation and localises reasoning. A single keyword controls this aspect. We support both internal and external locking schemes for data (Section 3–4).
- We formalise our system in the context of the language κ (pronounced kappa), state the key invariants of our system (safe aliasing, data-race freedom, strong encapsulation, thread-affinity and partial determinism) and prove them sound (Section 6–7).

The full proofs, dynamic semantics and a few longer code examples can be found in the accompanying technical report [15].

2 Problem Overview

Object-oriented programs construct graphs of objects whose entangled structure can make seemingly simple operations hard to reason about. For example, the behaviour of the following program (adapted from [28]) manipulating two counters `c1` and `c2` depends on

whether `c1` and `c2` may alias, which may only be true for some runs of the program.

```
assert c1.value() == 42; c1.inc(); c2.inc(); assert c1.value() == 43;
```

If `c1` and `c2` *always alias*, we may reason about the sequential case, but if `c2.inc()` is performed by *another thread*, the behaviour is affected by the scheduling of `c2.inc()`, and whether `inc()` itself is thread-safe. While aliasing is possible without sharing across threads, sharing across threads is not possible without aliasing. With this in mind, we move on to three case studies to discuss some of the challenges facing concurrent object-oriented programming.

2.1 Case Study: Simple Counters

To achieve thread-safety for a counter implemented in Java we can make the `inc()` method synchronised to ensure only one thread at a time can execute it. While this might seem straightforward, there are at least three problems with this approach:

1. Additional lock and unlock instructions for each increment will be inserted regardless of whether they are necessary or not – synchronising an unaliased object is a waste.
2. Making the object thread-safe does not help protect an instance from being shared, which might have correctness implications (*e.g.,* non-determinism due to concurrent accesses).
3. Unless the `value()` method is also synchronised, concurrent calls to `inc()` and `value()` may lead to a data-race, which might lead to a perception of lost increments.

In 1. and 2., the underlying problem is distinguishing objects shared across threads from thread-local objects as only the former needs synchronisation. Using two different classes for shared and unshared counters are possible, but leads to code duplication. Furthermore, if a counter is shared indirectly, *i.e.,* there is only one counter but its containing object is shared, the necessary concurrency control might be in the container. Establishing and maintaining such a "guarded-by property" warrants tool support.

In 3., the underlying problem is the absence of machinery for statically checking that all accesses to data are sufficiently protected. This might not be easy, for example, excluding data-races in methods inherited from a super class that encapsulates its locking behaviour.

2.2 Case Study: Data Parallelism and Task Parallelism

The counter exemplifies concurrent programming which deals with asynchronous behaviour and orchestration of operations on shared objects. In contrast, parallelism is about optimisation with the goal of improving some aspect of performance.

Consider performing the operations f_1 and f_2 on all elements in a collection E. A data parallel approach might apply $f_1(f_2(e))$ in parallel to all $e \in E$. In contrast, a task parallel approach might execute $f_1(e_1); \ldots; f_1(e_n)$ and $f_2(e_1); \ldots; f_2(e_n)$ as two parallel tasks.

Both forms of parallelism requires proper alias management to determine whether $f_1(e_i)$ and $f_2(e_j)$ may safely execute in parallel. When $i = j$, we must determine what parts of an object's interface might be used concurrently. When $i \neq j$, we must reason about the possible overlapping states of (the different) elements e_i and e_j. Furthermore, unless $f_1(e)$ (or $f_1(f_2(e))$) is safe to execute in parallel on the same object, we must exclude the possibility that E contains duplicate references to the same object.

If f_1 and f_2 only perform reads, any combination is trivially safe. However, correctly categorising methods as accessors or mutators manually can be tricky, especially if mutation happens deep down inside a nested object structure, and a method which may logically

only read might perform mutating operations under the hood for optimisation, telemetry, etc. Extending the categorisation of methods to include mutation of disjoint parts further complicates this task. Further, as software evolves, a method's categorisation might need to be revisited, even as a result of a non-local change (*e.g.,* in a superclass).

2.3 Case Study: Vector vs. ArrayList in Java

As a final case study, consider the `ArrayList` and `Vector` classes from the Java API. While both implement a list with comparable interfaces, vectors are thread-safe whereas array lists are not. There are several consequences of this design:

1. Vector objects lock individual operations. This requires multiple acquires and releases for compound operations (*e.g.,* when using an external iterator to access multiple elements).
2. The reliance on Java objects' built-in synchronisation excludes concurrent reads.
3. Just like the counter above, even thread-local vectors pay the price of synchronisation.

As a result, `ArrayList` is commonly favoured over `Vector` despite the fact that this requires locks to be acquired correctly for each use, rather than once if built into the data structure.

A lock that allows multiple concurrent reads (a readers–writer lock) would allow both vectors and array lists to be used efficiently and safely in parallel. This distinction adds an extra dimension of locking and requires categorising methods as accessors/mutators.

Summary. The examples above illustrate a number of challenges facing programmers doing concurrent and parallel programming in object-oriented languages. In summary:

- Code that needs synchronisation for data-race freedom is indistinguishable from code that does not. The same holds for code correctly achieving non-interference.
- Conservatively adding locks to all data structure definitions or all uses of a data structure hurts performance.
- Using locks to exclude conflicting concurrent accesses is non-trivial and requires reasoning about aliasing and program-wide sharing of data structures. The same reasoning is required for partitioning a data structure across multiple threads for parallel operations on disjoint parts, or specifying read-only operations.
- The need for concurrency control varies across different usage scenarios. Building concurrency control into data structures generates overhead or leads to code duplication (one thread-safe version and one which is not). Leaving concurrency control in the hands of clients instead opens up for under-synchronisation and concurrency bugs.
- The need for alias control varies across different usage scenarios. At times, thread-locality or even stronger aliasing restrictions are desirable, for example to avoid locks or non-determinism, or to unlock compiler optimisations or simplify verification. At other times, sharing is required. The sharing requirements of a single object could even vary over time.

We now describe our reference capability system which addresses all of these problems.

3 Capabilities for Concurrency Control

Our starting point for this work is to unify references and capabilities. A capability is a handle to a resource—a part of or an entire object or aggregate (an object containing other objects). A capability exposes a set of operations, which can be used to access its resource

without possibility of data-races. Granting and revoking capabilities corresponds to creating and destroying aliases. Capabilities' *modes* controls how they may be shared across threads:

Exclusive capabilities denote resources that are exclusive to one thread so that accesses are trivially free from any interference from other threads. There are two exclusive modes: `linear`, used for resources to which there is only a single handle in the program, and `thread`, which allows sharing, but only within one single thread. `linear` capabilities must be fully transferred from one thread in order to be used by another thread.

Safe capabilities denote resources that can be arbitrarily shared (*e.g.,* across multiple threads). There are two safe modes: `locked`, causing operations to be implicitly guarded by locks, and `read` which do not allow causing or directly observing mutation. Safe capabilities guarantee data-race freedom.

Subordinate capabilities (the mode `subordinate`) denote resources that are encapsulated inside some object and therefore inherit its protection against data-races or interference. Subordinate capabilities are similar to `rep` or `owner` in ownership types [16].

Unsafe capabilities (the mode `unsafe`) denote arbitrarily shared resources which are unsafe to use concurrently without some means of concurrency control. Accesses to unsafe capabilities must be wrapped in explicit locking instructions.

Linear capabilities impose transfer semantics on assignment. We adopt destructive reads [27] here for simplicity. This means that reading a variable holding a linear capability has the side-effect of updating it with `null`. Methods in `locked` capabilities automatically get acquire and release instructions, providing *per-method* atomicity. For `unsafe` capabilities locking must be done manually, providing *scoped* atomicity (the duration of the lock). Although straightforward, for simplicity we do not allow manual locking of `locked` in this presentation.

Types are compositions of one or more capabilities (*cf.,* Section 3.3) and expose the union of their operations. The modes of the capabilities in a type control how resources of that type can be aliased. The compositional aspect of our capabilities is an important difference from normal type qualifiers (*cf., e.g.,* [24]), as accessing different parts of an object through different capabilities in the same type gives different properties.

Exclusive and read capabilities guarantee *non-interference* and enable deterministic parallelism. Safe capabilities guarantee the absence of *data-races*, *i.e.,* concurrent write–write or read–write operations to the same memory location, but do not exclude *race-conditions*, *e.g.,* two threads competing for the same lock. This means that programs will be thread-safe, only one thread can hold the lock, but not necessarily deterministic—the order in which competing threads acquire a lock is controlled by factors external to the program. This also means that capabilities using locks do not exclude the possibility of deadlocks.

3.1 Capability = Trait + Mode

We present our capabilities system through κ, a Java-like language that uses traits [39] in place of inheritance for object-oriented reuse. A κ capability corresponds to a trait with some required fields, provided methods, and a mode. For the reader not familiar with traits, a trait can be thought of as an abstract class whose *fields* are abstract and must be provided by a concrete subclass—see Figure 4 for a code example of traits and classes. An important property of κ is that an *implementer of a trait can assume freedom from data-races or interference*, which enables sequential reasoning for all data that the trait *owns*, (its subordinate capabilities), plus reachable exclusive capabilities. A trait's mode controls how data-race freedom or non-interference is guaranteed. For example, prohibiting aliases to cross thread boundaries or inserting locks at compile-time in its methods.

The mode of a trait is either *manifest* or must be given wherever the trait is included by a class. A manifest mode is part of the declaration of the trait, meaning the trait defines a single capability. As an example of this, consider the capability **read** `Comparable` which provides compare methods to a class which do not mutate the underlying object. Traits without manifest modes can be used to construct different capabilities, *e.g.,* a trait `Cell` might be used to form both a **locked** `Cell` and a **linear** `Cell` when included in different classes, with different constraints on aliasing of its instances.

As a consequence of this design $\mathcal{K}$ allows the same set of traits to be used to construct classes tailored to different concurrency scenarios, thus contributing to trait-based reuse.

3.2 Dominating and Subordinate Capabilities

Building a data structure from linear capabilities gives *strong encapsulation*: subobjects of the data structure are not aliased from outside. However, linearity imposes a tree-shaped structure on data. Subordinate capabilities instead provide strong encapsulation by forbidding aliases from outside an aggregate to objects within the aggregate. Inside an aggregate, subordinate capabilities may be aliased freely, enabling any graph structure to be expressed.

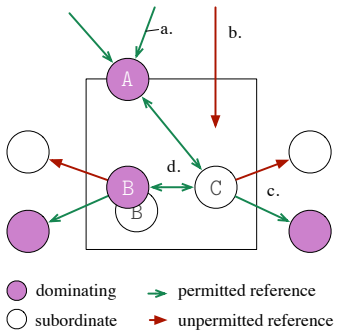
The capabilities **linear**, **thread**, **locked** and **unsafe** are *dominating capabilities* that enclose subordinate capabilities in a statically enforced way. Domination means that all *direct* accesses to objects inside an aggregate from outside are disallowed, making the dominator a single point of entry into an aggregate. As a consequence, any operation on an object inside an aggregate must be triggered by a method call on its dominating capability (directly or indirectly). This means that subordinate objects inherit the concurrency control of their dominator. Subordinate capabilities dominated by a **thread** capability inherit its thread-locality; subordinate capabilities dominated by a **locked** capability enjoys protection of its lock, etc. An implementation of a linked list with **subordinate** links inside a dominating list head guarantees that only a single thread at a time can mutate the links, while still allowing arbitrary internal aliasing inside of the data structure (*e.g.,* doubly-linked, circular).

Figure 1 shows encapsulation in $\mathcal{K}$ from dominating and subordinate capabilities. To enforce the encapsulation of subordinate objects, a subordinate capability (`B` and `C`) may not be returned from or passed outside of its dominating capability (`A`). There is no hierarchical decomposition of the heap (*cf.,* [16]) and no notion of transitive ownership. However, compositions (*cf.* Section 3.3) of dominating and subordinate capabilities (`B`) create nested aggregates, *i.e.,* entire aggregates strongly encapsulated inside another. Pointers to external capabilities must all be to dominating capabilities. Thus, objects inside `B` can refer to `A`, but not to `C`.

3.3 Flat and Nested Composition

As usual in a trait-based system, $\mathcal{K}$ constructs classes by composing traits, or rather capabilities. There are two forms of composition: disjunction ($\oplus$) and conjunction ($\otimes$). If `A` and `B` are capabilities, their disjunction `A` $\oplus$ `B` provides the union of the methods of `A` and `B` and requires the union of their field requirements. Their conjunction `A` $\otimes$ `B` does the same, but is only well-formed if `A` and `B` do not share mutable state which is not protected by concurrency control. This means that `A` $\otimes$ `B` allows `A` and `B` to be used in parallel. Figure 2 shows the composition constraints of disjunction and conjunction pictorially.

We use the term *flat composition* to mean disjunction or conjunction. When employing parametric polymorphism a form of *nested composition* appears. The nested capability `A<B>` exposes that `A` contains zero or more `B`'s at the type level, allowing type-level operations on



a. Disallowed if `A` is **`linear`**. If `A` is **`thread`**, aliases must come from same thread.

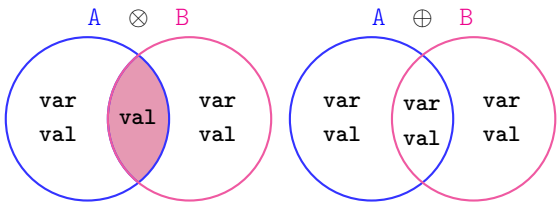
b. References from outside an aggregate to its inside are not permitted.

c. References from inside an aggregate to its outside are permitted if the target is a dominating capability.

d. References inside an aggregate are allowed.

The box encloses the subordinate capabilities of `A`. Note that `B` is a composition of a subordinate *and* a dominating capability (*cf.* Section 3.3), denoted by the two circles. All dominating capabilities have their own boxes (as shown for `A`), *e.g.,* `B` has a box nested inside of `A`'s box, inaccessible to `A` (*cf.,* b.).

Figure 1 Encapsulation: dominating and subordinate capabilities.



Explanation: **`val`** fields are "final", **`var`** fields are mutable. Intersections denote variables shared between (*i.e.,* **`require`**'d by) both capabilities. Types of fields in the filled intersection must be safe, *i.e.,* **`locked`** or **`read`**. Fields of **`subordinate`** type in a conjunction also must not alias.

Figure 2 Permitted sharing of fields and state across two capabilities `A` and `B` in a composite.

the composite capability. (This presentation uses a "dumbed down" version of parametric polymorphism using concrete types in place of polymorphic parameters for simplicity.)

A composite capability inherits all properties and constraints of its sub-capabilities. Linear capabilities must not be aliased at all. Subordinate capabilities must not leak outside their dominator. Consequently, a type which is both **`subordinate`** and **`linear`** is both a dominator (may encapsulate state) and a subordinate (is encapsulated), may not escape its enclosing aggregate and has transfer semantics when assigned (*cf.,* `B` in Figure 1).

Composition affects locking. A disjunction of two locked capabilities `A` ⊕ `B` will be protected by a single lock. A conjunction `A` ⊗ `B` of locked capabilities can use different locks for `A` and `B`, allowing each disjoint part to be locked separately. Furthermore, compositions of **`read`** and **`locked`** capabilities can be mapped to readers–writer locks.

An important invariant in κ is that all aliases are safe with respect to data-races or interference and can be used to the full extent of their types. If an alias can be created, any use of it will not lead to a bad race, either because it employs some kind of locking, because all aliases are read-only, or because the referenced object is exclusive to a particular thread.

4 Creating and Destroying Aliases = Concurrency Control

As aliasing is a prerequisite to sharing objects across possibly parallel computations, creating and destroying aliases is key to enabling parallelism while still guaranteeing race freedom in κ. Alias restrictions allows statically checkable non-interference, *i.e.,* without dynamic concurrency control (*e.g.,* locking). Programs that require objects that are aliased across threads must employ locks or avoid mutation.

Subordinate and thread-local capabilities may only be aliased from within certain contexts. Read, locked and unsafe capabilities have no alias restrictions. Finally, linear capabilities are

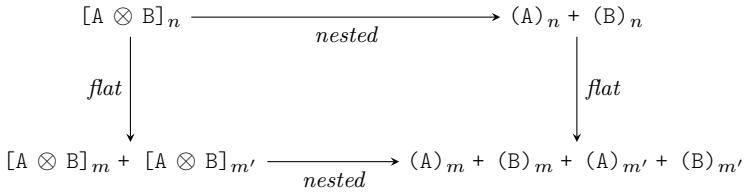


Figure 3 Flat and nested unpacking, using arrays as an analogy. $\texttt{[A} \otimes \texttt{B]}_n$, an array of length n containing composite capabilities $\texttt{A} \otimes \texttt{B}$ can be thought of as a matrix with rows as elements and whose columns are the elements' subparts, `A` and `B`. The matrix can be unpacked by rows (flat) or by columns (nested). Flat unpacking splits the array into subarrays of length m and m' such that $n = m + m'$. Nested unpacking requires that the containing object is not mutable, denoted by turning arrays into tuples, $\texttt{(A)}_m$. These compose in any order producing the same result.

alias-free. The following sections explore how linear types can be manipulated to create and destroy aliases (granting and revoking capabilities) while enjoying non-interference.

4.1 Packing and Unpacking

Conjunctions describe objects constructed from parts that can be manipulated in parallel without internal races. Unpacking breaks an object up into its sub-parts. A variable `c` with a handle to an instance of a class `C`, where **`class`** `C = A` $\otimes$ `B`, can be unpacked into two handles with types `A` and `B` using the `+` operator: **`var`** `a:A + b:B = c`, nullifying `c` in the process.

Unpacking a disjunction is unsafe (and therefore disallowed) since its building blocks can share mutable state not mediated by concurrency control. The dual of unpacking is packing, which re-assembles an object by revoking (nullifying) its sub-capabilities: **`var`** `c:C = a + b`.

The packing and unpacking above is *flat*. Using an array as analogy, flat unpacking takes an array $\texttt{[A]}_n$ with indexes $[0, n)$ and turns it into two disjoint equi-typed sub-arrays with indexes $[0, m)$ and $[m, n)$ where $m \leq n$. $\mathcal{K}$ also allows *nested unpacking*, which in the array analogy means that $\texttt{[A} \otimes \texttt{B]}_n$ can be unpacked into two tuples $\texttt{(A)}_n$ and $\texttt{(B)}_n$ with the same length and indexes. Turning the array into tuples, *i.e.,* immutable arrays of mutable values, is necessary as the aliases could otherwise be used to perform conflicting operations, *e.g.,* updating the `B`-part of element i in one thread and nullifying element i in another thread.

While safe capabilities can always be shared, unpacking allows a linear capability to be split into several aliases that can safely be used concurrently. When restoring the original capability through packing, there may be no residual aliases. We implement this here by preserving linearity in the unpacked capabilities. Figure 3 shows flat and nested unpacking and how they combine and commute. Section 5 shows how unpacking can be used to implement both data parallelism and task parallelism.

In this paper, we only consider packing and unpacking as operations at the level of types: their purpose is to statically guarantee non-interference, not construct new objects from other parts. Thus, packing can be efficiently compiled into an identity check or removed by a compiler provided that handles do not escape the scope in which they were unpacked.

4.2 Bounding Capabilities to the Stack

Linearity is often overly restrictive since it prevents even short-lived aliases that do not break any invariants. To remedy this, $\mathcal{K}$ employs *borrowing* [8]: temporarily relaxing linearity as long as the original capability is not accessible in the same scope, and all aliases are destroyed at the end of the scope. Borrowed capabilities in $\mathcal{K}$ are *stack-bound*, denoted by a

```
class Pair = (linear Fst ⊗ linear Snd) ⊕ linear Swap { var fst:int; var snd:int; }

trait Fst {
  require var fst:int;

  def setFst(i:int) : void {
    this.fst = i;
  }
  def getFst() : int {
    this.fst;
  } }

trait Snd {
  require var snd:int;

  def setSnd(i:int) : void {
    this.snd = i;
  }
  def getSnd() : int {
    this.snd;
  } }

trait Swap {
  require var fst:int;
  require var snd:int;

  def swap() : void {
    var tmp:int = this.fst;
    this.fst = this.snd;
    this.snd = tmp;
  } }
```

Figure 4 A pair class constructed from capabilities, `Fst`, `Snd` and `Swap`.

type wrapper `S()`. For example, `S(linear Cell)` denotes a capability which is identical to the `linear Cell` capability except that it may not be stored in a field, and thus is revoked once the scope exits. κ supports two forms of borrowing:

Forward Borrowing A `linear` capability in a stack variable can be converted into a stack-bound capability for a certain scope, destructively read and then safely reinstated at the end of the scope. This allows *e.g.,* passing a linear capability as an argument to a method, reinstating it on return. In conjunction with the borrowing it may optionally be converted to a `thread`, allowing it to be freely aliased until reinstated.

Reverse Borrowing A method of a `linear` capability may non-destructively read and return a stack-bound alias of a field of `linear` type. This allows linear elements of a data structure to be accessed without removing them, which is safe as long as the capability holding the field is not accessed during borrowing. To prevent multiple reverse borrowings of the same value (which would break linearity), the returned value may not be stored in fields or local variables but must be used immediately, *e.g.,* as an argument to a method call.

Borrowing simplifies programming with linear capabilities as it removes the need to explicitly consume and reinstate values when aliasing is benign, avoiding unnecessary memory writes. See Section 5 for an example of both forward and reverse borrowing in action.

4.3 Forgetting and Recovering Sub-Capabilities

Unpacking a disjunction is unsafe as its building blocks may have direct access to the same state without any concurrency control. As an example, consider the simple `Pair` class created from the capabilities `Fst`, `Snd` and `Swap` shown in Figure 4.

If we could unpack the pair, it would allow `fst` and `snd` to be updated independently. However, this is unsafe in the presence of the `Swap` capability, which accesses both fields. For example, the result of calling `swap()` concurrently with `setFst()` depends on the timing of the threads. A crude solution is simply upcasting `Pair` to `linear Fst ⊗ linear Snd`. This *forgets* the `Swap` capability and enables unpacking—but as a consequence `Swap` is lost forever.

To facilitate recovering a more specific type, κ provides a means to temporarily stash capabilities inside a *jail* which precludes their use except for recovering a composite type:

```
var p:Pair = ...;
var j:J(Pair|Fst ⊗ Snd) + k:(Fst ⊗ Snd) = p; // (1)
var f:Fst + s:Snd = k; // flat unpacking
... // use f and s freely
p = j + (f + s); // flat packing, twice, and getting out of jail (2)
```

At (1), the type of `j`, `J(Pair|Fst ⊗ Snd)`, denotes a jail storing a `Pair` which is unusable (the interface of a jailed capability is empty) until it is unlocked by providing the `Fst ⊗ Snd` capability of the corresponding resource as key. Thus `j` serves as a witness to the existence of the full `Pair` capability, including `Swap`. At (2), we recover `k` from `f` and `s`, nullifying both variables. We use the resulting value to open the jail `j` and store the result in `p`. As for packing, checking whether a key "fits" at run-time (*i.e.,* if `f` and `s` are aliases of the jail) is a simple pointer identity check, which could often be optimised away using escape analysis.

5 Applying Capabilities to the Case Studies in Section 2.1–2.3

Simple Counters. This example demonstrated the problem of distinguishing objects shared across threads from thread-local or unaliased objects objects, and pointed at the trickiness of locking correctly. In $\mathcal{K}$, a counter might be described as a simple trait `Counter`:

```
trait Counter {
  require var cnt : int;
  def inc() : void { this.cnt = this.cnt + 1; }
  def value() : int { return this.cnt; } }
```

To get a capability from the trait, what is missing is to add the mode declaration, which controls aliasing and sharing across threads. Out of the six possible mode annotations, five are allowed for the `Counter` trait:

linear A globally unaliased counter.
thread A thread-local counter. It can be aliased, but aliases cannot cross into other threads.
locked A counter protected by a lock, sharable across threads.
subordinate This type denotes a counter nested inside another object from which it cannot escape. It thus inherits data-race freedom or non-interference of the enclosing object.
unsafe A sharable, unprotected counter that requires the client to perform synchronisation at use-site: `c.inc()` will not compile unless wrapped inside a synchronisation block, which changes the type of `c` from **`unsafe`** to **`locked`**.

Using the mode **`read`** would denote a read-only counter, sharable across threads. Assigning this mode to the trait is rejected by the compiler because of the mutable `cnt` field.

Modes communicate how counters may be aliased: not at all, by a single thread, or across threads. In the latter case modes also communicate how concurrent accesses are made safe: by locks, by only allowing reads (not applicable here), by relying on some containing object or by delegating responsibility to the client.

Differently synchronised counters can be defined almost without code duplication, *e.g.*:

```
class LocalCounter = thread Counter { var cnt:int; }
class SharedCounter = locked Counter { var cnt:int; }
```

Data/Task Parallelism. This example demonstrated the need for reasoning about aliasing in order to determine what parts of an interface can be safely accessed concurrently.

A binary tree can be constructed as the conjunction of capabilities giving access to the left and right subtrees and the current element (full code in the technical report [15]).

```
class Tree<T> = linear Left<T> ⊗ linear Right<T> ⊗ linear Element<T>
```

We employ nesting to show that the tree contains capabilities of type `T`, the type of the element value held by the `Element` capability. The conjunction allows parallel operations on subparts of a tree and requires that parts do not overlap, modulo safe capabilities. Since the

tree type must be treated linearly, the fact that the `Left` and `Right` subtrees do not overlap follows from the requirement that `Left` and `Right` manipulate fields of different names.

To perform data-parallel operations on a tree, we can construct a recursive procedure that takes a tree, splits it into its separate components and operates on them in parallel.

```
def foreach(t:S(Tree<T>), f:T → T) : void {
  var l:S(linear Left<T>) + r:S(linear Right<T>) + e:S(linear Element<T>) = t; // 0
  finish {
    async { foreach(l.getLeft(), f); } // 1
    async { foreach(r.getRight(), f); } // 1
    e.apply(f); } } // 2
```

At (0) the splitting implicitly consumes the original tree capability. At (1) we recurse on the left and right subtrees. At (2) we pass the function argument `f` to the element capability to be performed on its `T`-typed value. For simplicity, we omit the check for whether `l` or `r` is `null`. The implementation requires a tree to be constructed from linear building blocks to guarantee that no parts of the tree are ever shared across multiple threads. `T` does not need to be linear.

This code illustrates both forward and reverse borrowing. The tree argument to `foreach()` is forward borrowed and stack-bound, which is why there is no need to pack `l`, `r` and `e` to recover `t`—`t` is still accessible at the call-site, where it was buried [8] during the call.

Calls to `getLeft()` and `getRight()` return two reversely borrowed linear values (of type `S(Tree<T>)`) which we can pass as arguments to the recursive calls. Hence, all trees manipulated by this code will be stack-bound. If we remove the stack-boundedness, `foreach()` may not update the subtrees in-place, and must recover and return `t` at the end, reminiscent of functional programming. This would cause lines marked (1) to change thus:

```
async { l.setLeft( foreach( l.getLeft(), f ) ); }
async { l.setRight( foreach( l.getRight(), f ) ); }
```

which allows *replacing* the tree as opposed to updating it, plus a return: `return l + r + e`.

We may extend the `Tree` type with a disjunction on a capability `Visit` which provides a read-only view of the entire tree. Elements may not be swapped for other elements, but modified if `T` allows it. This allows multiple threads to access the same tree in parallel provided that `Left`, `Right` and `Element` are temporarily forgotten.

```
class Tree<T> = read Visit<T> ⊕ (linear Left<T> ⊗ linear Right<T> ⊗ linear Element<T>)
```

Let the type of our tree be `Tree<A ⊗ B>` for linear capabilities `A` and `B`. Turning this capability into `Visit<A ⊗ B>` is possible by forgetting every other capability in the tree type. While read-only capabilities can be aliased freely, creating multiple aliases typed `Visit<A ⊗ B>` would provide multiple paths to supposedly linear `A ⊗ B` capabilities. Composition must thus adhere to all alias restrictions in the composite capability, just like flat composition. Therefore, `Visit<A ⊗ B>` is a linear capability. Unpacking however allows us to turn `Visit<A ⊗ B>` into two handles typed `Visit<A>` and `Visit<B>`, which preserves linearity across all paths. This allows us to specify a task-parallel operation which implements column-based access:

```
def map(t:S(Tree<A⊗B>), f:S(A)→void, g:S(B)→void) : void{
  var ta:S(read Visit<A>) + tb:S(read Visit<B>) = t; // 3
  finish {
    async { ta.preorder(f); } // 4
    async { tb.preorder(g); } } }// 4
```

In this code we create two immutable views of the spine of the tree using `Visit` and then proceed to apply `f` and `g` to all elements of the tree in parallel. At (3) the rest of the capabilities of `Tree` are forgotten. If we wanted to restore them after the parallel operations we would jail them at (3) and restore them after (4).

While the data-parallel version is more scalable than the task-parallel version, there may be cases when the latter is preferred. Further, their combination is possible in either order—apply `f` and `g` in parallel to each element at (2) above, or start by unpacking the tree into multiple immutable trees and then process the sub-elements in parallel in each tree, equivalent to calling a version of `foreach` instead of `preorder` at (4) (*cf.,* Figure 3).

Vector vs. ArrayList in Java. This example demonstrated that building synchronisation into a data structure can cause too much overhead and destroy parallelism. In κ, a list might be described using capabilities (full code in technical report, [15]):

- `Add_Del` for adding and removing elements
- `Get` for looking up elements

`Add_Del` might be split into two capabilities allowing for more flexibility, for example granting a client only the ability to add elements but not delete them. As the two capabilities operate on some shared state (the links), their combination must be a disjunction: `Add_Del` $\oplus$ `Get`.

To express the difference between the array list and vector, we would write

```
class ArrayList = unsafe Add_Del ⊕ unsafe Get // Needs external synchronisation
class Vector = locked Add_Del ⊕ locked Get // Has synchronisation built in
```

Specifying use of readers–writer locks to access an object is straightforward and allows sharing a list across threads for reading, causing concurrent write operations to block:

```
class ArrayList = unsafe Add_Del ⊕ read Get
class Vector = locked Add_Del ⊕ read Get
```

The use of **`unsafe`** in the definition of the array list class pushes the synchronisation from within the called methods to the outside, *e.g.,* calling `list.add(element)` we must first take a (write-)lock on `list`. Requiring external synchronisation also allows acquiring, holding and releasing a lock once to perform several operations, like an iteration, without fear of interleaving accesses from elsewhere.

The type **`thread`** `Add_Del` $\oplus$ **`read`** `Get` denotes a list confined to its creating thread. The type **`linear`** `Add_Del` $\oplus$ **`read`** `Get` denotes a list that can mediate between being mutated from one alias or read-only from several aliases. This type is similar to a readers–writer lock, except relying on alias restrictions instead of locks (*cf.,* [9]), removing locking overhead. The ability to reuse traits for different concurrency scenarios is an important contribution of κ.

Concluding Remarks for Section 3–5

Linear and thread-local capabilities give *non-interference* by restricting aliases to a single thread. Locked and unsafe capabilities can be shared across threads and employ locks at declaration-site or at use-site to *avoid data-races*. Read capabilities can be shared across threads and do not allow causing or directly witnessing mutation. When a read capability is extracted from a linear composite, no mutating aliases exist, guaranteeing *non-interference*. When extracted from a locked composite, locks are used to guarantee *data-race freedom*.

The assignment of modes to traits at inclusion site allow a single definition to be reused across multiple concurrency scenarios. Composition captures how different parts of an object's interface interact and defines the safe aliasing of an object.

Subordinate capabilities inherit the protection of their enclosing dominating capabilities. Thus, operations on encapsulated objects are atomic in κ, in the sense that all side-effects of a method call on an aggregate are made visible to other threads atomically. Operating atomically on several objects which are not encapsulated in the same aggregate is possible by

$$
\begin{array}{rcll}
P & ::= & Cds\ Tds\ e & \textit{(Program)} \\
Cd & ::= & \texttt{class}\ \texttt{C} = \texttt{K}\ \{\ Fds\ \} & \textit{(Class definition)} \\
Fd & ::= & mod\ f : \texttt{t} & \textit{(Field definition)} \\
mod & ::= & \texttt{var} \mid \texttt{val} & \textit{(Mutable and immutable fields)} \\
\texttt{K} & ::= & k\ \texttt{T} \mid k\ \mathbf{I} \mid \texttt{K}\langle\texttt{K}\rangle \mid (\texttt{K} \odot \texttt{K}) & \textit{(Capabilities and composition)} \\
\odot & ::= & \otimes \mid \oplus & \textit{(Conjunction and disjunction)} \\
Td & ::= & k\ \texttt{trait}\ \texttt{T}\langle t\rangle\ \{Rs\ Mds\ \} \mid \texttt{trait}\ \texttt{T}\langle t\rangle\ \{Rs\ Mds\ \} & \textit{(Trait definition)} \\
R & ::= & \texttt{require}\ Fd & \textit{(Field requirement)} \\
Md & ::= & \texttt{def}\ m(x : t) : t\ \{\ e\ \} & \textit{(Method definition)} \\
\texttt{e} & ::= & v \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{pack}\ x = y\ +\ z\ \texttt{in}\ e \mid \texttt{unpack}\ x\ +\ y = z\ \texttt{in}\ e \mid x.m(e) \mid x \mid x.f & \\
 & \mid & x.f = e \mid \texttt{new}\ \texttt{C} \mid \texttt{consume}\ x \mid \texttt{consume}\ x.f \mid (t)\ e \mid \texttt{sync}\ x\ \texttt{as}\ y\ \{\ e\ \}; e & \\
 & \mid & \texttt{bound}\ x\ \{\ e\ \}; e \mid \texttt{finish}\ \{\ \texttt{async}\ \{\ e\ \}\ \texttt{async}\ \{\ e\ \}\ \}; e & \textit{(Expression)} \\
v & ::= & \texttt{null} & \textit{(Literal)} \\
t & ::= & \texttt{K} \mid \texttt{C} \mid B(\texttt{K}) & \textit{(Type)} \\
B & ::= & \texttt{J}_{\texttt{K}} \mid \texttt{S} & \textit{(“Boxed” types, i.e., jailed or stack-bound)} \\
k & ::= & \texttt{linear} \mid \texttt{locked} \mid \texttt{read} \mid \texttt{safe} \mid \texttt{subordinate} \mid \texttt{thread} \mid \texttt{unsafe} & \textit{(Modes)}
\end{array}
$$

Figure 5 Syntax of κ. T is a trait name; $\mathbf{I}$ is the incapability; C is a class name; m is a method name; f is a field name; x, y are variable names, including **this**. $Ds ::= D_1, \ldots, D_n$ for $D \in \{Cd, Td, Fd, R, Md\}$.

locking them together using nested synchronisation (for **unsafe** capabilities) or by structuring a call-chain on **locked** capabilities.

Invariantly, all well-typed aliases can coexist without risking data-races. The type system guarantees that all accesses to an object will either be exclusive or only perform operations that cannot clash with any other possible concurrent operations to the same object.

6 Formalising κ

We formalise the static semantics of κ. We define a flattening translation into a language without traits, κ_{F}, whose static and dynamic semantics is found in the accompanying technical report [15]. κ_{F} is a simple object-oriented language with structured parallelism and locking, that uses classes and interfaces which are oblivious to the existence of κ capabilities. The translation from κ to κ_{F} inserts locking and unlocking operations when translating **locked** capabilities and conjunctions of **locked** and **read** capabilities. The locks are reentrant readers–writer locks controlling access to parts of objects. Other locking schemes are possible.

The syntax of κ is shown in Figure 5. We make a few simplifications, none of which are critical for the soundness of the approach:

1. We use let-bindings and explicit pack/unpack constructs. Targets of method calls must be stack variables. Aliasing stack-bounds requires a method-call indirection.
2. We consider finish/async parallelism rather than unstructured creation of threads.
3. Classes only contain fields and no methods.
4. We omit the treatment of constructors. Fields are initialised with **null** on instantiation.
5. We use objects to model higher-order functions and omit these from the formalism.
6. Only a single method parameter and a single nested type are supported.

We introduce a **safe** capability, which abstracts **read** and **locked** to allow mode subtyping. The **safe** mode is only allowed in types, not in declarations. The *incapability* type $\mathbf{I}$ does not contain any fields or methods and simply allows holding a reference to an object.

$\boxed{\vdash P : t \quad \vdash Td \quad \Gamma \vdash Td \quad \vdash Cd}$ *(Well-formedness top-level declarations)*

WF-PROGRAM
$$\frac{\forall\ Cd \in Cds\ .\ \vdash Cd \quad \forall\ Td \in Tds\ .\ \vdash Td \quad \epsilon \vdash e : t}{\vdash Cds\ Tds\ e : t}$$

WF-T-TRAIT
$$\frac{\vdash \mathbf{subord}\ \mathbf{trait}\ \mathtt{T}\langle t\rangle\ \{\ Rs\ Mds\ \}}{\vdash \mathbf{trait}\ \mathtt{T}\langle t\rangle\ \{\ Rs\ Mds\ \}}$$

WF-T-TRAIT-MFST
$$\frac{\rho : t \vdash k\ \mathbf{trait}\ \mathtt{T}\ \{\ Rs\ Mds\ \}}{\vdash k\ \mathbf{trait}\ \mathtt{T}\langle t\rangle\ \{\ Rs\ Mds\ \}}$$

WF-T-INNARDS
$$\frac{k \neq \mathbf{safe} \quad \Gamma, \mathbf{this} : k\,\mathtt{T} \vdash Rs \quad \Gamma, \mathbf{this} : k\,\mathtt{T} \vdash Mds}{\Gamma \vdash k\ \mathbf{trait}\ \mathtt{T}\ \{\ Rs\ Mds\ \}}$$

WF-CLASS
$$\frac{\vdash \mathtt{K} \quad \forall\ Fd \in Fds\ .\ \mathbf{this} : \mathtt{K} \vdash Fd \quad \forall\ \mathbf{var}\ f : t_1 \in \mathbf{fields}\,(\mathtt{K})\ .\ \exists\ \mathbf{var}\ f : t_2 \in Fds\ .\ t_2 \equiv t_1 \quad \forall\ \mathbf{val}\ f : t_1 \in \mathbf{fields}\,(\mathtt{K})\ .\ \exists\ \mathbf{var}\ f : t_2 \in Fds\ .\ t_2 <: t_1}{\vdash \mathbf{class}\ C = \mathtt{K}\ \{\ Fds\ \}}$$

$\boxed{\Gamma \vdash R, Rs \quad \Gamma \vdash Mds, Md \quad \Gamma \vdash Fd \quad \vdash \Gamma}$ *(Well-formed body parts)*

WF-REQ-FD
$$\frac{\Gamma \vdash Fd}{\Gamma \vdash \mathbf{require}\ Fd}$$

WF-REQ-FDS
$$\frac{\Gamma \vdash R \quad \Gamma \vdash Rs}{\Gamma \vdash Rs, R}$$

WF-MDS
$$\frac{\Gamma \vdash Md \quad \Gamma \vdash Mds}{\Gamma \vdash Mds, Md}$$

WF-M-TRAIT
$$\frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \mathbf{def}\ m(x : t_1) : t_2\ \{\ e\ \}}$$

WF-FD-NST
$$\frac{\vdash \Gamma \quad \Gamma(\rho) = \mathtt{K}}{\Gamma \vdash \mathbf{val}\ f : \mathtt{K}}$$

WF-FD
$$\frac{\Gamma(\mathbf{this}) = \mathtt{K} \quad \vdash t \quad t \neq \mathbf{S}(\_) \quad \vdash \mathtt{K} \quad \mathbf{thread}(t) \Rightarrow \mathbf{thread}(\mathtt{K}) \quad \mathbf{read}(\mathtt{K}) \Rightarrow (mod \equiv \mathbf{val} \wedge \mathbf{safe}(t))}{\Gamma \vdash mod\ f : t}$$

ENV-E
$$\frac{}{\vdash \epsilon}$$

ENV-VAR
$$\frac{\vdash \Gamma \quad \vdash t \quad x \notin \mathbf{dom}(\Gamma)}{\vdash \Gamma, x : t}$$

Figure 6 Well-formed declarations. $\Gamma ::= \epsilon \mid \Gamma, x : t$, ($x$ incl. ρ).

Our main technical result is the proof that a κ_{F} program translated from a well-typed κ program enjoys safe aliasing and strong encapsulation (*cf.* Section 7.2) in a way that implies thread-safety (*cf.* Section 7.3). We verify our definition of thread-safety by proving that it implies data-race freedom and, when certain capabilities are excluded, also non-interference (*cf.* Section 7.3).

Helper Predicates and Functions. The functions **fields**, **vals**, **vars** and **msigs** return a map from names to types or method signatures. We use helper predicates of the form k(K) to assess whether a capability K has mode k. The predicates **linear**, **subord**(inate) and **unsafe** hold if there exists *at least one* sub-capability in K of that mode. The predicates **read**(K) and **encaps**(K) hold if *all* sub-capabilities in K are **read** or **subordinate**, respectively. **locked**(K) holds if one or more sub-capabilities are locked, and the remainder **safe**.

6.1 Well-Formed κ Programs (Figure 6)

A well-formed program consists of classes, traits, and an initial expression (WF-PROGRAM). Traits without manifest mode are type-checked as if they were subordinate (WF-T-TRAIT). To reduce the number of rules, we require all traits to have exactly one nested capability (a concrete type "parameter"), and use T as shorthand for T<**I**>, where **I** is the empty capability. A trait is well-formed if its field requirements and methods are well-typed given the self-type of the current trait and the nested type. The latter is tracked by the special variable ρ which

$\boxed{\vdash t}$ *(well-formedness of types)* $\boxed{Fd_1 \odot Fd_2}$ *(sharing fields across traits)*

T-CLASS
$$\frac{\textbf{class}\ C = \mathtt{K}\,\{\ \_\ \} \in P}{\vdash C}$$

T-BOXED
$$\frac{\vdash \mathtt{K}}{\vdash B(\mathtt{K})}$$

T-NESTING
$$\frac{\vdash \mathtt{K}_1 \quad \vdash \mathtt{K}_2}{\vdash \mathtt{K}_1\langle\mathtt{K}_2\rangle}$$

C-SHARABLE
$$\frac{\textbf{safe}(t) \vee \textbf{unsafe}(t)}{\textbf{val}\, f : t \otimes \textbf{val}\, f : t}$$

T-TRAIT-MFST
$$\frac{k\ \textbf{trait}\ \mathtt{T}\langle\mathtt{K}\rangle\ \{\ \_\_\_\ \} \in P}{\vdash k\,\mathtt{T}}$$

T-TRAIT
$$\frac{k \equiv \textbf{read} \Rightarrow \textbf{this} : \textbf{read}\,\mathtt{T} \vdash Rs \quad \textbf{trait}\ \mathtt{T}\langle\mathtt{K}\rangle\ \{\ Rs\ \_\ \} \in P}{\vdash k\,\mathtt{T}}$$

C-VAR-VAL
$$\frac{t_1 <: t_2}{\textbf{var}\, f : t_1 \oplus \textbf{val}\, f : t_2}$$

C-VAL-VAL
$$\frac{\vdash \mathtt{K}_1 \otimes \mathtt{K}_2 \quad \odot\ =\ \otimes \Rightarrow \neg\,\textbf{subord}(\mathtt{K}_1 \otimes \mathtt{K}_2)}{\textbf{val}\, f : \mathtt{K}_1 \odot \textbf{val}\, f : \mathtt{K}_2}$$

T-COMPOSITION
$$\frac{\vdash \mathtt{K}_1 \qquad \vdash \mathtt{K}_2 \quad \forall\ Fd_1 \in \textbf{fields}\,(\mathtt{K}_1), Fd_2 \in \textbf{fields}\,(\mathtt{K}_2)\ .\ Fd_1 \odot Fd_2 \quad \textbf{wfRegions}\,(\mathtt{K}_1, \mathtt{K}_2) \qquad \textbf{wfRegions}\,(\mathtt{K}_2, \mathtt{K}_1)}{\vdash \mathtt{K}_1 \odot \mathtt{K}_2}$$

T-I
$$\frac{}{\vdash k\,\mathbf{I}}$$

C-DISJOINT
$$\frac{f_1 \neq f_2}{mod_1\, f_1 : t_1 \odot mod_2\, f_2 : t_2}$$

T-REGIONS
$$\frac{\forall\ \mathtt{K}_1 \otimes \mathtt{K}_2 \in \mathtt{K}\ .\ mod_1\, f_1 : t_1 \in \textbf{fields}\,(\mathtt{K}_1) \wedge mod_2\, f_2 : t_2 \in \textbf{fields}\,(\mathtt{K}_2) \wedge f_1 \neq f_2 \wedge \textbf{subord}(t_1) \wedge \textbf{subord}(t_2) \Rightarrow \neg\,(f_1 \in \textbf{fields}\,(\mathtt{K}') \wedge f_2 \in \textbf{fields}\,(\mathtt{K}'))}{\textbf{wfRegions}\,(\mathtt{K}, \mathtt{K}')}$$

C-DISJUNCTION
$$\frac{}{mod_1\, f : t \oplus mod_2\, f : t}$$

Figure 7 Well-formed types. Conjunctions and disjunctions of traits are governed by the rules for overlapping fields. For simplicity, we omit (C-VAL-VAR) which is isomorphic with (C-VAR-VAL).

may not appear anywhere in the program source (WF-T-TRAIT-MFST). Fields are either mutable (`var`) or stable (`val`). We assume that names of classes and traits are unique in a program and the names of fields and methods are unique in classes and traits.

A well-formed class consists of well-typed `var` fields that satisfy the requirements from its traits, and a defined equivalence to a well-formed composite capability. We allow covariance for `val` fields (WF-CLASS). Only immutable fields holding `safe` capabilities are allowed in `read` capabilities (WF-REQ-*,WF-FD), unless the type of the field is exposed through nesting (WF-FD-NST). Fields may not store stack-bound capabilities and fields holding thread-local values are only allowed if the containing object is also thread-local (WF-FD).

6.2 Well-Formed Types (Figure 7)

Capabilities corresponding to traits with manifest modes are trivially well-formed (T-TRAIT-MFST). Traits without a manifest mode can be given any mode (T-TRAIT). Well-formed `read` capabilities may only contain `safe` val fields. The empty capability **I** can be given any mode (T-I). Composing capabilities with **I** thus affects the mode of the composite, but not the interface (*cf.,* Section 6.3).

Two well-formed capabilities can form a nested capability type (T-NESTING). A composite capability is well-formed if its sub-capabilities are well-formed and their shared fields are composable (T-COMPOSITION). We also require that two subordinate fields appearing on opposing sides of a conjunction $\mathtt{K}_1 \otimes \mathtt{K}_2$ are not both accessible from some other trait $\mathtt{K}'$ in the same composite (T-REGIONS). Such a field would act as a channel that could be used to share subordinate state across the supposedly disjoint representations of $\mathtt{K}_1$ and $\mathtt{K}_2$.

The rules of the form $Fd_1 \odot Fd_2$ govern field overlaps between capabilities in a composite, where $\odot \in \{\otimes, \oplus\}$ denotes the composition of the capabilities containing the fields (*cf.*, Figure 2). Disjunctions may overlap freely (C-DISJUNCTION). Disjoint fields do not overlap (C-DISJOINT). If a field appearing on both sides of a composition is mutable on one side and immutable on the other, the mutable field's type must be more precise (C-VAR-VAL). An immutable field may appear on both sides of a composition only if its type is safe or unsafe (C-SHARABLE) or if the fields have types whose conjunction is well-formed (C-VAL-VAL). If the sharing capabilities are conjunctive, the field must not be subordinate.

6.3 Type Equivalence, Packing and Subtyping (Figure 8)

Class names are aliases for composite capabilities (T-EQ-CLASS-TRAIT). The order of the operands in composition of *a single kind* does not matter (T-EQ-COMMUTATIVE) and (T-EQ-ASSOCIATIVE). Equivalent types in jail or bound to the stack are still equivalent (T-EQ-BOXED). A **read** with a nested conjunction can be unpacked into two **read** capabilities with nested capabilities from the unpacked conjunction (T-EQ-NESTING). Incapabilities can be added and removed from a type as long as modes are preserved (T-EQ-I).

Jailing allows creating a conjunction from a disjunction (T-JAIL). The full capability that can be unlocked from the jail is written as a subscript K. Jailing requires that all modes on the composite are preserved by the extracted capability, modulo the rules $k_{<:}$ for unpacking to be sound. For example, **locked** A $\oplus$ **subord** B denotes a (partially) subordinate capability that is strongly encapsulated inside an aggregate. If we are allowed to forget the subordinate mode of the type, the locked sub-capability could be leaked. We employ (T-EQ-I) to this end. For example, we can compose **subord** **I** with **locked** A $\oplus$ **subord** B and then apply (T-JAIL) to extract **locked** A $\oplus$ **subord** **I** which satisfies mode preservation.

Subtyping is structural on capabilities. Subtyping must preserve modes, or encapsulation, domination or exclusivity could be lost. The rules (T-SUB-*) allow **locked** and **read** to be abstracted by the **safe** mode.

6.4 Well-Typed Expressions (Figure 9)

Packing & Unpacking. The rules (E-PACK) and (E-UNPACK) govern the packing and unpacking of capabilities. They rely on the rules (T-JAIL) and (T-PACK) in Figure 8 which allow introducing aliases to discrete capabilities of a conjunction and turning a disjunction into a conjunction by jailing the overlapping parts.

Linearity. **linear** capabilities are destructively read to maintain alias freedom and allow ownership transfer (E-CONS-VAR,E-CONS-FD). Method calls do not destroy **linear** receivers as an object's **this** cannot be consumed. Thus, linear capabilities are externally unique [17].

Finish–Async and Sync. Parallelism in $\mathcal{K}$ is modelled using a scoped finish/async construct (abbreviated as **f** and **a** respectively) (E-FJ). A finish–async forks two parallel computations and waits until they have both completed. Forking a larger number of threads can be simulated using nested finish/async blocks. For simplicity we do not allow async blocks outside of a finish block, but extending the system to support unstructured parallelism or active objects is possible. We employ a frame rule that guarantees that no variable is visible in both asyncs, and that subordinate objects are only accessible to the first of the asyncs, (FRAME). This models the current thread running the first async (with access to the current **this**), and another thread running the second async block.

$\boxed{t_1 \equiv t_2}$ *(type equivalence)*

T-EQ-CLASS-TRAIT
$$\frac{\textbf{class}\ C = \mathtt{K}\,\{\ \_\ \} \in P}{C \equiv \mathtt{K}}$$

T-EQ-COMMUTATIVE
$$\frac{}{\mathtt{K}_1 \odot \mathtt{K}_2 \equiv \mathtt{K}_2 \odot \mathtt{K}_1}$$

T-EQ-ASSOCIATIVE
$$\frac{}{(\mathtt{K}_1 \odot \mathtt{K}_2) \odot \mathtt{K}_3 \equiv \mathtt{K}_1 \odot (\mathtt{K}_2 \odot \mathtt{K}_3)}$$

T-EQ-BOXED
$$\frac{\mathtt{K}_1 \equiv \mathtt{K}_2}{B(\mathtt{K}_1) \equiv B(\mathtt{K}_2)}$$

T-EQ-NESTING
$$\frac{\textbf{read}(\mathtt{K}_1)}{\mathtt{K}_1\langle \mathtt{K}_2 \otimes \mathtt{K}_3\rangle \equiv \mathtt{K}_1\langle \mathtt{K}_2\rangle \otimes \mathtt{K}_1\langle \mathtt{K}_3\rangle}$$

T-EQ-I
$$\frac{k(\mathtt{K})}{\mathtt{K} \odot k\,\mathbf{I} \equiv \mathtt{K}}$$

T-EQ-TRANS
$$\frac{\mathtt{K}_1 \equiv \mathtt{K}_2 \qquad \mathtt{K}_2 \equiv \mathtt{K}_3}{\mathtt{K}_1 \equiv \mathtt{K}_3}$$

$\boxed{t_1 \rightleftharpoons t_2 \otimes t_3}$ *(packing and unpacking)*

T-PACK
$$\frac{\mathtt{K}_1 \equiv \mathtt{K}_2 \otimes \mathtt{K}_3 \qquad \textbf{subord}(\mathtt{K}_2) \Leftrightarrow \textbf{subord}(\mathtt{K}_3)}{\mathtt{K}_1 \rightleftharpoons \mathtt{K}_2 \otimes \mathtt{K}_3}$$

T-JAIL
$$\frac{\mathtt{K}_1 \equiv \mathtt{K}_2 \oplus \mathtt{K}_3 \qquad \forall\ k\ .\ k(\mathtt{K}_1) \Rightarrow k(\mathtt{K}_3)}{\mathtt{K}_1 \rightleftharpoons \mathbf{J}_{\mathtt{K}_1}(\mathtt{K}_2) \otimes \mathtt{K}_3}$$

T-PACK-BOUND
$$\frac{\mathtt{K}_1 \rightleftharpoons \mathtt{K}_2 \otimes \mathtt{K}_3}{\mathbf{S}(\mathtt{K}_1) \rightleftharpoons \mathbf{S}(\mathtt{K}_2) \otimes \mathbf{S}(\mathtt{K}_3)}$$

$\boxed{t_1 <: t_2 \quad k_{<:}(\mathtt{K})}$ *(subtyping and "submoding")*

T-SUB-STRUCTURAL
$$\frac{\forall\ k\ .\ k(\mathtt{K}_1 \odot \mathtt{K}_2) \Rightarrow k_{<:}(\mathtt{K}_1)}{\mathtt{K}_1 \odot \mathtt{K}_2 <: \mathtt{K}_1}$$

T-SUB-BOXED
$$\frac{\mathtt{K}_1 <: \mathtt{K}_2}{B(\mathtt{K}_1) <: B(\mathtt{K}_2)}$$

T-SUB-EQ
$$\frac{t_1 \equiv t_2 \qquad t_2 <: t_3}{t_1 <: t_3}$$

T-SUB-ID
$$\frac{}{t <: t}$$

T-SUB-K
$$\frac{k(\mathtt{K})}{k_{<:}(\mathtt{K})}$$

T-SUB-RD
$$\frac{\textbf{safe}(\mathtt{K})}{\textbf{read}_{<:}(\mathtt{K})}$$

T-SUB-LOCK
$$\frac{\textbf{safe}(\mathtt{K})}{\textbf{locked}_{<:}(\mathtt{K})}$$

Figure 8 Type equivalence, packing/unpacking, subtyping.

FRAME
$$\frac{\textbf{dom}\,(\Gamma_2) \cap \textbf{dom}\,(\Gamma_3) \equiv \emptyset \qquad \nexists\ x : t \in \Gamma_3\ .\ (\textbf{subord}(t) \vee \textbf{thread}(t)) \qquad \forall\ x\ .\ (\Gamma_2(x) = t \Rightarrow \Gamma_1(x) = t) \wedge (\Gamma_3(x) = t \Rightarrow \Gamma_1(x) = t)}{\Gamma_1 = \Gamma_2 + \Gamma_3}$$

The `sync` keyword temporarily converts an `unsafe` (and therefore unusable) capability into a `locked` capability, acquiring and releasing locks at the entry and exit of e_1 (E-SYNC).

Borrowing. Forward borrowing allows turning capabilities into stack-bound capabilities non-destructively, possibly relaxing a `linear` to a `thread` and allows splitting reads (E-FORWARD). The helper predicate **boundable**($\mathtt{K}_1$, $\mathtt{K}_2$) holds in either of three cases:

1. $\mathtt{K}_1 = \mathtt{K}_2$
2. $\mathtt{K}_1[\texttt{linear} \mapsto \texttt{thread}] = \mathtt{K}_2$ (relaxing linearity to thread-affinity)
3. $\mathtt{K}_1 = \mathtt{K}_2 \odot \mathtt{K}_3$ s.t. $\mathtt{K}_3$ is neither `subordinate` nor `thread`, and all capabilities in $\mathtt{K}_2$ are `read`.

The last case allows relaxing alias restrictions for stack-bound `read` capabilities which enables mediating from single writer to multiple readers across multiple threads without dynamic concurrency control for a clearly defined scope.

Reverse borrowing allows non-destructively reading a `linear` capability into a stack-bound value (E-REVERSE). Since only one value can be returned by a method, multiple reverse borrowing of the same field in the same method is innocuous. Non-linear capabilities never need to be reverse borrowed as they can always be returned normally without consuming.

Self Typing. Modulo traits with manifest modes, `this` inside a trait is always subordinate (WF-T-TRAIT). This reflects the fact that on the inside of a capability, exclusive access of a

$\boxed{\Gamma \vdash e : t}$ *(expression typing)*

E-UPCAST
$$\frac{\Gamma \vdash e : t_2 \qquad t_2 <: t_1}{\Gamma \vdash (t_1)e : t_1}$$

E-NEW
$$\frac{\vdash \Gamma \qquad \textbf{class}\ C = \mathtt{K}\{\_\} \in P}{\Gamma \vdash \textbf{new}\ C : \mathtt{K}}$$

E-NULL
$$\frac{\vdash \Gamma \qquad \vdash t}{\Gamma \vdash \textbf{null} : t}$$

E-LET
$$\frac{\Gamma \vdash e_1 : t_1 \qquad t_1 \neq \mathbf{S}(\_) \qquad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : t_2}$$

E-UNPACK
$$\frac{\Gamma \vdash z : t_1 \qquad t_1 \rightleftharpoons t_2 \otimes t_3 \qquad \Gamma, x : t_2, y : t_3 \vdash e : t_4}{\Gamma \vdash \textbf{unpack}\ x + y = z\ \textbf{in}\ e : t_4}$$

E-PACK
$$\frac{\Gamma \vdash y : t_1 \qquad \Gamma \vdash z : t_2 \qquad \textbf{linear}(t_1) \qquad \textbf{linear}(t_2) \qquad t_3 \rightleftharpoons t_1 \otimes t_2 \qquad \Gamma, x : t_3 \vdash e : t_4}{\Gamma \vdash \textbf{pack}\ x = y + z\ \textbf{in}\ e : t_4}$$

E-SELECT
$$\frac{\Gamma \vdash \textbf{this} : t_1 \qquad \textbf{fields}\,(t_1)(f) = t_2 \qquad \neg\,\textbf{linear}(t_2)}{\Gamma \vdash \textbf{this}.f : t_2}$$

E-CONS-FD
$$\frac{\Gamma \vdash \textbf{this} : t_1 \qquad \textbf{vars}\,(t_1)(f) = t_2}{\Gamma \vdash \textbf{consume}\,\textbf{this}.f : t_2}$$

E-UPDATE
$$\frac{\Gamma \vdash \textbf{this} : t_1 \qquad \textbf{vars}\,(t_1)(f) = t_2 \qquad \Gamma \vdash e : t_2}{\Gamma \vdash \textbf{this}.f = e : t_2}$$

E-VAR
$$\frac{\vdash \Gamma \qquad \Gamma(x) = t \qquad \neg\,\textbf{linear}(t)}{\Gamma \vdash x : t}$$

E-CONS-VAR
$$\frac{\vdash \Gamma \qquad x \neq \textbf{this} \qquad \Gamma(x) = t}{\Gamma \vdash \textbf{consume}\ x : t}$$

E-CALL
$$\frac{\textbf{linear}(t_1) \Rightarrow x \notin \textbf{freeVars}\,(e) \qquad \Gamma(x) = t_1 \qquad \neg\,\textbf{unsafe}(t_1) \qquad \textbf{msigs}\,(t_1)(m) = z : t_2 \to t_3 \qquad \Gamma \vdash e : t_2 \qquad (\textbf{subord}(t_2) \vee \textbf{subord}(t_3)) \Rightarrow \textbf{encaps}\,(t_1) \vee x \equiv \textbf{this}}{\Gamma \vdash x.m(e) : t_3}$$

E-FJ
$$\frac{\Gamma = \Gamma_1 + \Gamma_2 \qquad \Gamma \vdash e : t \qquad \Gamma_1 \vdash e_1 : \_ \qquad \Gamma_2 \vdash e_2 : \_}{\Gamma \vdash \mathbf{f}\,\{\ \mathbf{a}\ \{\ e_1\ \}\ \mathbf{a}\ \{\ e_2\ \}\ \};\, e : t}$$

E-SYNC
$$\frac{\Gamma \vdash x : \textbf{unsafe}\,\mathtt{T} \qquad \Gamma, y : \mathbf{S}(\textbf{locked}\,\mathtt{T}) \vdash e_1 : \_ \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash \textbf{sync}\ x\ \textbf{as}\ y\ \{\ e_1\ \};\, e_2 : t}$$

E-REVERSE
$$\frac{\Gamma \vdash \textbf{this} : t \qquad \textbf{linear}(t) \qquad \textbf{fields}\,(t)(f) = \mathtt{K}}{\Gamma \vdash \textbf{this}.f : \mathbf{S}(\mathtt{K})}$$

E-FORWARD
$$\frac{\textbf{boundable}\,(\mathtt{K}_1, \mathtt{K}_2) \qquad \Gamma, x : \mathbf{S}(\mathtt{K}_2) \vdash e_1 : \_ \qquad \Gamma, x : \mathtt{K}_1 \vdash e_2 : t}{\Gamma, x : \mathtt{K}_1 \vdash \textbf{bound}\ x\ \{\ e_1\ \};\, e_2 : t}$$

Figure 9 Typing of expressions. Note that all fields are "private".

single thread is already guaranteed (*e.g.,* because the accessing thread was forced to acquire a lock to enter the object, or place a linear entry point in a stack variable, which is analogous).

Viewing `this` as subordinate allows an object to be aliased freely from *inside* its own enclosure, including objects of linear capabilities. Thus, linear capabilities in κ are externally unique [17]. Traits with manifest modes have more information about themselves internally.

7 Meta-Theoretic Evaluation

In this section, we describe the key invariants of our capabilities in a well-typed κ program. Due to lack of space, proofs have been relegated to a technical report [15]. This section aims to explain the key properties, and sketching why they hold.

7.1 κ_{F} and the Dynamic Semantics of κ Programs

To execute κ programs, they are translated into a simpler language, κ_{F}, with classes and interfaces without traits or capabilities. The semantics of κ_{F} is straightforward, and only the most relevant details are presented here.

Flattening of traits [39] is performed similar to other trait-based languages: A class is translated by copying the methods from its traits, traits are translated into interfaces with the equivalent signature. For each composition $\mathtt{K}_1 \odot \mathtt{K}_2$, an interface is created extending the interfaces corresponding to $\mathtt{K}_1$ and $\mathtt{K}_2$. This preserves the same subtyping rules as in $\mathcal{K}$. For each $\mathcal{K}$-capability in a translated program, there is a single corresponding interface. Because of this one-to-one mapping, we can easily recover $\mathcal{K}$ types from a $\mathcal{K}_\mathrm{F}$ program, which we use extensively in our proofs.

From field overlaps, we infer a set of regions for each class, and insert lock and unlock instructions at the start and end of methods to acquire and release the lock for the region touched by the method. Read locks are acquired in methods in read capabilities if they overlap with a locked or unsafe capability. Methods in locked capabilities acquire write locks. All locks are reentrant. Well-formedness of $\mathcal{K}_\mathrm{F}$ configurations requires that no two threads can hold the same writer lock, and that all locks held by a thread are also taken in the objects themselves. This assures mutual exclusion in all parts of a program that use locks.

A well-formed $\mathcal{K}$ program will always translate into a well-formed $\mathcal{K}_\mathrm{F}$ program. This is easy to prove as most type rules for $\mathcal{K}_\mathrm{F}$ are subsumed by the $\mathcal{K}$ type rules. Thus, by proving type soundness (progress and preservation) of $\mathcal{K}_\mathrm{F}$, we show that a translated $\mathcal{K}$ program will never get stuck (modulo deadlocks, which we distinguish from unsound stuck states).

$\mathcal{K}_\mathrm{F}$ imposes no restrictions on aliasing nor does it provide any guarantees about race freedom. However, $\mathcal{K}_\mathrm{F}$ programs translated from well-formed $\mathcal{K}$ programs will always be data-race free. As $\mathcal{K}_\mathrm{F}$ itself provides no data-race guarantees, the invariants given by $\mathcal{K}$ are defined independent of well-formed configurations of $\mathcal{K}_\mathrm{F}$ (see Section 7.2 and Section 7.3).

We have a fully mechanised specification of $\mathcal{K}_\mathrm{F}$ in Coq, including a proof of type soundness, but not data-race freedom. In our hand-written proof of data-race freedom we also extend $\mathcal{K}_\mathrm{F}$ with means of tracking types and stack-boundedness of values. This has no effect on the execution or typechecking of $\mathcal{K}_\mathrm{F}$ programs and is thus excluded from the mechanised version [15]. Specifying all of $\mathcal{K}$ in Coq is future work.

7.2 Aliasing and Encapsulation

This section details the invariants on aliasing and encapsulation in well-typed $\mathcal{K}$ programs.

Safe Aliasing. One of the main technical results of this work is the proof that $\mathcal{K}$ programs enjoy *safe aliasing*, *i.e.,* two paths to the same mutable field are local to the same thread, or protected by the same lock which must be acquired before access. Informally, aliasing is safe if the following is true for all aliases x, y on the stack or heap:

- x and y have composable types, meaning they point to different parts of the same object, modulo safe `val` fields, corresponding to $\vdash t_x \otimes t_y$ in $\mathcal{K}$ (*cf.,* Figure 7).
- x and y are protected, *i.e.,* read-only aliases, or safe aliases that use locks internally, or unsafe aliases whose accesses must be wrapped in locks.
- x and y are both local to the same thread, corresponding to the `thread` capabilities of $\mathcal{K}$.
- x and y both have subordinate types, meaning that any thread accessing them must currently have exclusive access to their dominator.
- If x or y is `linear`, at least one of the aliases must be stack-bound to prevent introducing aliases of linear values on the heap.
- If one of the aliases is stack-bound, the origins of the borrowed value must be buried, to prevent multiple accessible references to the same linear value.

Proof. Part of the proof of thread-safety [15].

Strong Encapsulation of Subordinate Capabilities. Another invariant preserved by κ programs is that references in fields of subordinate type point to objects dominated by the dominator of the current enclosure. This is what grants subordinate capabilities strong encapsulation, similar to ownership types [16] and external uniqueness [17].

At run-time in κ_{F}, instances know the identity of their dominator. This identity is invariant, even under ownership transfer, because transfer operates on linear capabilities and instances of classes without a subordinate capability are their own dominators.

Let $\rightarrow$ denote "refers to" and $\iota.\texttt{dom}$ denote the dominator for an object with id ι in the heap H. Now, $\forall\, \iota, \iota' \in dom(H)$, $\iota \rightarrow \iota'$ s.t. $\iota \neq \iota'$, either one of the following holds:

1. $\iota'.\texttt{dom} = \iota.\texttt{dom}$ (a pointer between subordinates in the same enclosure)
2. $\iota'.\texttt{dom} = \iota$ (a dominator pointing to one of its subordinates)
3. $\iota.\texttt{dom} = \iota'$ (a subordinate pointing to its dominator)

or ι' is a top-level object, *i.e.,* $\iota'.\texttt{dom} = \iota'$.

Proof. Part of the proof of thread-safety [15].

7.3 Data-Race Freedom and Non-Interference.

This section describes the invariants of κ for concurrent and parallel programming.

Thread-Safety. Safe aliasing and the encapsulation guarantees mentioned above are both part of a bigger notion of a *thread-safe* (**TS**) configuration. A configuration is thread-safe if no two possible reductions can cause interference—if a possible reduction of a configuration has one thread writing to a field, there cannot be another reduction of the same configuration where the same field is read or written to by another thread.

In addition to safe aliasing, a number of constraints apply to the elements of thread-safe configurations that deal specifically with aliasing across threads:

- references in fields of subordinate type point to objects dominated by the dominator of the current enclosure;
- values of type **thread** were created by the thread that can access them;
- all local variables of **subordinate** type are dominated by the closest dominating `this` on the current stack;
- if a stack-bound `linear` value aliases a value on the heap, the value on the heap is effectively buried, *i.e.,* the only path to it is rooted on the stack;
- all accesses to values not wrapped in locks are `linear`, `thread`, `subord` or `read`.

Having defined thread-safety, we then prove that the initial configuration is **TS** and that evaluation preserves this property in a program translated from κ to κ_{F}.

▶ Preservation of Thread-Safety. In a well-formed program translated from κ, if a thread-safe configuration *cfg* can step to *cfg*′, then *cfg*′ is also thread-safe.

$$\forall\, \Gamma,\ cfg,\ cfg'.\\ \Gamma \vdash \mathbf{TS}(cfg) \wedge cfg \hookrightarrow cfg' \Rightarrow \exists \Gamma'.\Gamma' \vdash \mathbf{TS}(cfg') \wedge \Gamma \subseteq \Gamma'$$

Proof. We prove thread-safety by induction over the thread structure [15]. The proof is similar in structure to the type preservation proof of κ_{F} (and relies on type preservation to find Γ').

Data-Race Freedom. The dynamic semantics of κ_{F} tracks effect footprints in terms of reads ($\mathbf{rd}(\_)$) and writes ($\mathbf{wr}(\_)$) of object fields. This allows us to define and prove data-race freedom, which verifies that our notion of thread-safety is sound.

▶ Data-Race Freedom. If a safe configuration *cfg* steps to two different configurations causing effects Eff_1 and Eff_2 respectively, then these effects are non-conflicting:

$$\forall \Gamma,\ \mathit{cfg},\ \mathit{cfg}'\ \mathit{cfg}''\ .\\ \Gamma \vdash \mathbf{TS}(\mathit{cfg}) \wedge \mathit{cfg} \hookrightarrow^{\mathit{Eff}_1} \mathit{cfg}' \wedge \mathit{cfg} \hookrightarrow^{\mathit{Eff}_2} \mathit{cfg}'' \Rightarrow \mathit{Eff}_1 \# \mathit{Eff}_2 \vee \mathit{cfg}' = \mathit{cfg}''$$

where $\#$ denotes that two effects are disjoint or a read–read conflict:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathbf{rd}(l.f) & \# & \mathbf{rd}(l'.f') & \_(l.f) & \# & \mathbf{wr}(l'.f') \quad \text{iff} \quad l \neq l' \vee f \neq f' \\ \varepsilon & \# & \_ & \mathit{Eff}_1 & \# & \mathit{Eff}_2 \quad \text{iff} \quad \mathit{Eff}_2 \# \mathit{Eff}_1 \end{array}$$

Proof. The proof is straightforward and performed by case analysis on the thread structure, showing that any interference contradicts thread-safety [15].

Corollary: Non-Interference. Parallel κ expressions that do not use locked or unsafe capabilities are *free from interference and therefore deterministic*. The only way threads can affect each other is by writing shared mutable locations, which requires taking a lock. Without locked or unsafe capabilities there are no locks, meaning that any data that is shared between threads can only be read, and that no threads are ever blocked in their execution.

Corollary: Thread-Affinity of Thread Capabilities. Implied by **TS**, κ `thread` capabilities are thread-affine. Let $\mathit{creator}(\iota)$ return the id of the thread creating ι. From **TS** follows that in a thread *tid* with local variables V and expression e, $\forall x\,.\,V(x) = \iota \wedge\ \Gamma(x) = t \wedge\ \mathbf{thread}(t) \Rightarrow \mathit{creator}(\iota) = \mathit{tid}$ and $\forall \iota\,.\,\iota \in \mathbf{locations}(e) \wedge\ \Gamma(x) = t \wedge\ \mathbf{thread}(t) \Rightarrow \mathit{creator}(\iota) = \mathit{tid}$. Thus `thread` capabilities are only visible to their creating threads. The key elements in the type system are the $\mathbf{thread}(t) \Longrightarrow \mathbf{thread}(\mathrm{K})$ constraint in the κ rule (WF-FD) which restrict fields of type `thread` to only appear inside manifestly `thread` capabilities and the κ rule (FRAME) which does not allow `thread` capabilities to be visible in the second `async` of a `finish`.

8 Related Work

An original source of inspiration for this work was Boyland *et al.*'s "Capabilities for Sharing" which introduces a system of reference capabilities, such as immutability or ownership in a dynamic system, not amenable to static typing [10]. Similarly, κ brings together ideas from many different areas in a single system, but fully statically typed. To the best of our knowledge, the selection and integration of the features in κ are unique in an object setting:

1. Linear capabilities are similar to uniqueness [27, 34] or permissions [9, 42, 38] and enable ownership transfer [17].
2. Subordinate capabilities enable strong encapsulation similar to ownership types [16] or Universes [35, 36]. κ's combination of subordinate and dominating capabilities give arbitrary nesting, but nested aggregates may not refer to subordinate objects in their enclosing aggregate, nor does κ support incoming read-only references (owners-as-modifiers [35, 36])—both enable data-races.

3. Thread capabilities resemble thread-local heaps in Loci [43] and ownership for actors [18].
4. That linear capabilities view themselves as subordinate capabilities internally gives a form of external uniqueness [17, 26].
5. The combination of locked and read capabilities express readers–writer locks, with a compile-time guarantee that readers will not write.
6. The `safe` mode, abstracting over `read` and `locked`, avoids code duplication for traits that are agnostic to why objects are safe, similar to type qualifier generics [22, 45].
7. The combination of locked and subordinate capabilities empowers a single lock to range over an entire aggregate with a compile-time guarantee of correctness (*cf.,* owners-as-locks in *e.g.,* [16]), it also allows enforcing a crude form of lock ordering in combination with readers-writer locks by connecting lock order to nesting order (*cf.,* [7]).
8. Nested capabilities are essentially storable permissions [9, 42] but without breaking abstraction—the names of fields etc. of the object storing permissions can be kept secret.
9. The flat composition of capabilities and the packing/unpacking marries ideas from fractional permissions [9] with ownership and substructural types [13], similar to [29]. Composites of read and linear capabilities support mediation between readers and writers, using stack-bounding to identify where sharing starts and stops.

Ownership Systems. Aliasing of mutable state in object-oriented programming is a mature research field: categorisations of alias management techniques [28], ownership types [16], universe types [22, 36], external uniqueness [17], balloons [2], as well as multiple flavours of references [27, 4, 34, 3, 10, 8, 12] etc. (*cf.* [16] for a broad coverage of many aspects). Banning aliasing is usually abandoned in favour of alias control, which commonly prescribes a certain shape on the program [16, 22, 43, 18] possibly combined with an effect system to coordinate accesses to shared data across multiple program locations [19, 5, 14]. κ does not prescribe a certain topology for shared capabilities. With the shift to ubiquitous parallelism, ideas from these fields have been applied to the simplification of concurrent and parallel programming (*e.g.,* [7, 18, 26, 24, 37, 21]).

Abadi *et al.* [1] propose RaceFree Java where field declarations are associated with locks and an effect system tracks how locks are acquired and released. Classes can be parameterised with external locks. The combination of locked and subordinate capabilities in κ seem to be able to express the same, but without ghost variables or an effect system. Zhao [44] constructs a system similar to Abadi *et al.* [1] but based on fractional permissions. It uses method annotations with read/write effects and locks taken and also considers deadlocks through lock ordering, similar to [7].

The recent surge of interest in Mozilla's Rust language provides anecdotal evidence to the value of languages with data-race freedom built in. In Rust, values mediate between linear/mutable and sharable/immutable. Linear values use transfer semantics and Rust uses borrowing to simplify programming. Rust support flat unpacking of arrays, but not nested unpacking and not unpacking of other types. Rust does not have strong encapsulation meaning an aggregate's innards is not protected by the aggregate's single entry-point and no aliasing of mutable objects is allowed inside the aggregate.

Substructural Systems. κ is close in spirit to work by Caires and Seco [13] as well as work by Pottier *et al.* [38] on Mezzo, both in the context of ML-like languages. Caires and Seco formalise a fine-grained capability system for reasoning about interference caused by aliasing or concurrent accesses to aliased data with explicit synchronisation. There is no support for read-sharing or strong non-linear encapsulation.

Militão *et al.* use a substructural type system for specifying rely-guarantee protocols in a functional context [32]. Protocols capture the view of shared state from one particular alias—our capabilities are similar. Ownership transfer and recovery for linear values is supported.

The functional language Alms [40] explores the design space of practical programming with substructural types. Alms separates capabilities from references and operations that require a capability must have the capability passed in as an argument. Capabilities in Alms are lower level than in κ and can be used to express many of our capabilities. There is no unification of capabilities with building blocks like traits, or composition of capabilities.

Capability Systems and Permission Systems. Miller uses capabilities for access control and concurrency control in a distributed setting in the seminal E language [33], employing a more dynamic approach (than κ) with optional soft typing.

Mediation between different views of an object is similar to fractional permissions [9]. κ supports going from a single writer to multiple enumerable disjoint writers or readers, and in the case of readers to an unbounded number of stack-bound aliases via (E-FORWARD). Fractional permissions with nesting [11, 42] is similar to subordinate capabilities in allowing one permission to act as guard to another. These system allows turning an entire nested structure read-only. κ's subordinate capabilities are less restricted, but also not transferable. κ's `read` capabilities also provide abstraction as they allow fields remain private.

Bocchino's Deterministic Parallel Java [5] uses an effect system to guarantee deterministic parallelism for operations that have no overlapping writes. When the effect system is not enough, the user can annotate trusted operations as commuting. κ provides similar determinism guarantees when excluding locking capabilities, but resorts to locking (and non-determinism) rather than using unchecked annotations for more complex operations.

Clebsch *et al.* [20] use "deny capabilities" to provide safe sharing of objects between actors. Their capabilities always grant exclusive write access to entire objects, while κ's also allows accessing parts of an object, as well as permitting multiple parallel writers.

Westbrook *et al.* [42] formalise and implement a gradual extension to HabaneroJava, HJp, in the form of a permission system. Permissions in HJp always govern access to entire objects, and there is no notion of encapsulation modulo storing linear permission in fields which only supports tree-shaped data. When there is not enough permission information, dynamic checks are inserted which may fail, but which also allow unconstrained aliasing. Splitting a single write-permission into multiple read permissions is similar to `read` capabilities.

Chalice by Leino *et al.* [30] is a language for specification and verification of concurrent software that uses permissions to statically track aliasing. Since κ is concerned only with data-race freedom, it trades some of the more fine-grained control (*e.g.,* full method specification) for simplicity (*e.g.,* no need for manual permission tracking).

9 Discussion & Future Work

We currently require programmers to explicitly manage substructural operations manually through packing and unpacking and jails. Building simpler-to-use constructs on top of these is possible. For example, a combination of `bound` and `unpack` would remove the need for packing, at the cost of enforcing a nested structure to unpacking and packing. Employing inference to automate this to a large extent seems possible and is a direction for future work.

Implementation of κ is on-going in the actor-based language Encore. There, actors replace locks as a means of pessimistic concurrency control. Capabilities protect actors' state

while allowing ownership-transfer due to linear values. A larger case study evaluating the full expressiveness of κ is planned for future work.

Unstructured vs. Structured. We have purposely supported *unstructured* packing and unpacking of capabilities. This allows granting capabilities to other threads (possibly on other machines) without requiring the capabilities to be returned or tying their return to a particular local scope. This removes limitations inherent in effect systems (*e.g.,* [25, 19, 5]), which requires computations to be nested. Unstructured locking is important in some applications, for example to implement hand-over-hand locking, and is a possible direction for future work.

Revocation. We only consider "cooperative revocation", *i.e.,* there is no built-in mechanism to arbitrarily revoke a given capability. In the security setting from which the capability idea stems, this is a major concern but it makes less sense in our setting as revoking a capability from another thread at an unfortunate point in time might cause system-wide inconsistencies.

Other Capabilities. The only form of dynamic concurrency control considered in this paper is locks. In ongoing work, the set of modes are extended with `async` (objects are actors), `atomic` (objects use transactional memory) and `lockfree` (lock-free programming). In this richer setting, we aim to address more programmer-friendly forms of multi-object atomicity.

10 Conclusions

Creating and destroying aliases enables and constrains parallelism and is key to establishing data-race freedom and non-interference. By capturing how data is shared and accessed through modes, and by introducing a structured approach to creating and destroying aliases through the combination of capabilities that make up classes, data-race freedom can be guaranteed statically with or without dynamic concurrency control. κ's invariants are similar to what an effect system can give, but avoids complicated effect annotations which propagate through the program and constrain inheritance. Tracking modes in types provides machine-checked documentation about alias freedom and sharing which localises reasoning.

The unification of traits and capabilities allows a single trait to serve multiple concurrency scenarios, which extends trait-based reuse. It also simplifies programming as trait implementers may safely assume data-race freedom. Ultimately, κ brings together a broad spectrum of prior work in a unified system.

Acknowledgements. We are grateful for the comments from Sophia Drossopoulou, the SLURP reading group at Imperial College, Dave Clarke and the anonymous reviewers.

References

1 Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, March 2006. doi:10.1145/1119479.1119480.

2 Paulo Sérgio Almeida. *Control of Object Sharing in Programming Languages*. PhD thesis, Imperial College London, June 1998.

3 David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: a Dialect of Java Without Data Races. In *OOPSLA*, pages 382–400, 2000.

4 Henry G. Baker. 'Use-once' Variables and Linear Objects – Storage Management, Reflection and Multi-Threading. *ACM SIGPLAN Notices*, 30(1), January 1995.

5 Robert Bocchino. An Effect System and Language for Deterministic-By-Default Parallel Programming, 2010. PhD thesis, University of Illinois at Urbana-Champaign.

6 Shekhar Borkar and Andrew A Chien. The future of microprocessors. *CACM*, 54(5):67–77, 2011.

7 Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.

8 John Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, May 2001.

9 John Boyland. Checking interference with fractional permissions. In *SAS*, pages 55–72, 2003.

10 John Boyland, James Noble, and William Retert. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. ECOOP. Springer, 2001.

11 John Tang Boyland. Semantics of fractional permissions with nesting. *ACM Trans. Program. Lang. Syst.*, 32(6):22:1–22:33, August 2010. doi:10.1145/1749608.1749611.

12 John Tang Boyland and William Retert. Connecting Effects and Uniqueness with Adoption. In *POPL*, pages 283–295, 2005.

13 Luís Caires and João C. Seco. The Type Discipline of Behavioral Separation. POPL, 2013.

14 Nicholas R. Cameron, Sophia Drossopoulou, James Noble, and Matthew J. Smith. Multiple ownership. In *OOPSLA*, 2007.

15 E. Castegren and T. Wrigstad. Reference capabilities for trait based reuse and concurrency control. Technical Report 2016-007, 2016. Uppsala University.

16 Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership Types: A Survey. In *Aliasing in Object-Oriented Programming*, volume 7850 of *LNCS*. Springer, 2013.

17 Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *ECOOP*, 2003.

18 Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Johnsen. Minimal ownership for active objects. In *Programming Languages and Systems*, volume 5356 of *LNCS*. Springer, 2008.

19 David G. Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Type and Effect. In *OOPSLA*, pages 292–310, 2002.

20 Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *AGERE*, 2015.

21 David Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universe Types for Race Safety. In *VAMP*, 2007. URL: http://pubs.doc.ic.ac.uk/universes-races/.

22 Werner M. Dietl. *Universe Types: Topology, Encapsulation, Genericity, and Tools*. Ph.D., Department of Computer Science, ETH Zurich, December 2009.

23 M. Fähndrich and R. DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *PLDI*, pages 13–24, 2002.

24 Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and Reference Immutability for Safe Parallelism. In *OOPSLA*, pages 21–40, 2012.

25 Aaron Greenhouse and John Boyland. An object-oriented effects system. ECOOP, 1999.

26 Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *ECOOP*, 2010.

27 John Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *OOPSLA*, 1991.

28 John Hogg, Doug Lea, Alan Wills, Dennis de Champeaux, and Richard Holt. The Geneva Convention on the Treatment of Object Aliasing. *OOPS Messenger*, 3(2), April 1992.

29 Neelakantan R. Krishnaswami, Aaron Turon, Derek Dreyer, and Deepak Garg. Superficially Substructural Types. ICFP, New York, NY, USA, 2012. ACM.

30 K Rustan M Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*. 2009.

31 H. Levy, editor. *Capability Based Computer Systems*. Digital Press, 1984.

32 Filipe Militão, Jonathan Aldrich, and Luís Caires. Rely-guarantee protocols. In *ECOOP*, 2014.

33 Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

34 Naftaly Minsky. Towards alias-free pointers. In *ECOOP*, July 1996.

35 P. Müller and A. Poetzsch-Heffter. Universes: a type system for controlling representation exposure. Technical Report 263, 1999. Fernuniversität Hagen.

36 Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer, 2002.

37 Pratibha Permandla, Michael Roberson, and Chandrasekhar Boyapati. A type System for Preventing Data Races and Deadlocks in the Java Virtual Machine Language. In *LCTES*, pages 1–10, 2007.

38 Francois Pottier and Jonathan Protzenko. Programming with Permissions in Mezzo. In *ICFP*, pages 173–184, September 2013.

39 Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and AndrewP. Black. Traits: Composable units of behaviour. In *ECOOP*, 2003.

40 Jesse A. Tov. *Practical Programming with Substructural Types*. PhD thesis, Northeastern University, 2012.

41 Philip Wadler. Linear Types Can Change the World! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*. North Holland, 1990.

42 Edwin Westbrook, Jisheng Zhao, Zoran Budimli, and Vivek Sarkar. Practical permissions for race-free parallelism. In *ECOOP*, 2012.

43 Tobias Wrigstad, Filip Pizlo, Fadi Meawad, Lei Zhao, and Jan Vitek. Loci: Simple thread-locality for Java. In *ECOOP*, 2009.

44 Yang Zhao. *Concurrency Analysis Based On Fractional Permission System*. PhD thesis, University of Wisconsin – Milwaukee, 2007.

45 Yoav Zibin, Alex Potanin, Shay Artzi, et al. Object and reference immutability using Java generics. In *ESEC/FSE*. 2007.